Efficient d-ary Cuckoo Hashing at High Load Factors by Bubbling Up

Abstract

A d-ary cuckoo hash table is an open-addressed hash table that stores each key x in one of d random positions $h_1(x), h_2(x), \ldots, h_d(x)$. In the offline setting, where all items are given and keys need only be matched to locations, it is possible to support a load factor of $1 - \epsilon$ while using $d = \lceil \ln \epsilon^{-1} + o(1) \rceil$ hashes. The online setting, where keys are moved as new keys arrive sequentially, has the additional challenge of the time to insert new keys, and it has not been known whether one can use $d = O(\ln \epsilon^{-1})$ hashes to support $\mathrm{poly}(\epsilon^{-1})$ expected-time insertions.

In this paper, we introduce *bubble-up cuckoo hashing*, an implementation of d-ary cuckoo hashing that achieves all of the following properties simultaneously:

- uses $d = \lceil \ln \epsilon^{-1} + \alpha \rceil$ hash locations per item for an arbitrarily small positive constant α.

- achieves expected insertion time $O(\delta^{-1})$ for any insertion taking place at load factor $1 - \delta \leq 1 - \epsilon$.

- achieves expected positive query time $O(1)$, independent of d and ϵ.

The first two properties give an essentially optimal value of d without compromising insertion time. The third property is interesting even in the offline setting: it says that, even though *negative* queries must take time d, *positive* queries can actually be implemented in $O(1)$ expected time, even when d is large.

1   Introduction

In recent decades, cuckoo hashing has emerged as one of the most widely used studied hash table designs for both theory and practice (see, e.g., [2, 3, 7, 9, 11, 13, 16, 19, 23, 26, 28, 29, 31] as examples) . In its most basic form [29], cuckoo hashing is a technique for using two hash functions h_1, h_2 in order to place some number m of elements into an array of size n. The invariant that the hash table maintains is that, at any given moment, every element x is in one of positions $h_1(x), h_2(x) \in [n]$. Of course, it is not obvious that such an invariant is even possible. What Pagh and Rodler showed in their seminal 2001 paper [29] was that, so long as $m < n/2 - \Omega(n)$, then not only is the invariant possible (with probability $1 - O(1/n)$), but it is even possible to support insertions in $O(1)$ expected time.

The advantage of cuckoo hashing is its query time: every query completes in just 2 memory accesses. The main disadvantage – at least for the simplest version of the data structure – is its space efficiency: the hash table behaves well only if the load factor m/n is less than $1/2$. When the load factor surpasses $1/2$, with high probability one cannot place the elements into the hash table and maintain the invariant.

To support larger load factors, one can use $d > 2$ hash functions $h_1, h_2, \ldots, h_d$. This version of the data structure, known as *d-ary cuckoo hashing*, has been studied in both the offline (i.e., all of the elements are given to us up front) [8, 15, 18] and the online (i.e., the elements are inserted one by one) [13, 19, 16, 17, 3, 31, 10, 22] settings. Much of the work in the offline setting has focused on establishing the minimum value of d needed to support any given load factor $1 - \epsilon$. As $\epsilon \to 0$, the optimal value for d becomes $\lceil \ln \epsilon^{-1} + o(1) \rceil$, where the $o(1)$ term is a function of ϵ^{-1} [8, 15, 18].

The main challenge in the online setting is to support a small value for d while *also* supporting fast insertions as a function of ϵ^{-1}. Here, there are two high-level goals:

- **Goal 1:** Use a small value of d, ideally close to $\ln \epsilon^{-1}$.

- **Goal 2:** Support insertions in expected time close to $O(\epsilon^{-1})$.

On the upper-bound side, there have been two major steps forward so far: that one can support $d = \Theta(\log \epsilon^{-1})$ while offering expected insertion time $\epsilon^{-O(\log \log \epsilon^{-1})}$ [13]; and that one can support $d = \lceil \ln \epsilon^{-1} + o(1) \rceil$ while offering expected insertion time $f(\epsilon^{-1})$ for some unknown but potentially large function f [3]. Whether one can achieve an expected insertion time of the form $O(\epsilon^{-1})$, or even of the form $\text{poly}(\epsilon^{-1})$, while using $d = \ln \epsilon^{-1} + O(1)$, or even $d = O(\ln \epsilon^{-1})$, has remained open.

Finally, when it comes to *queries*, there is also a third goal that one might ask for:

- **Goal 3:** Support *positive* queries in *expected* time $O(1)$, independent of d and ϵ.

This final goal is a bit subtle. If we query an element x that is not present (this is a **negative** query), then we cannot help but spend d time on the query. If we query an element x that is present (this is a **positive** query), then the *worst-case* query time is d, but the *expected* query time could be smaller, since the query can stop as soon as it finds the element it's looking for. In principle, one could hope for an expected positive query time of $O(1)$.

It is not clear, even intuitively, whether one should expect these three goals to be compatible. One might imagine that there is tension between Goals 1 and 2, for example, that $O(\epsilon^{-1})$ insertions are possible when $d = (1 + \Omega(1)) \ln \epsilon^{-1}$ but not when $d = (1 + o(1)) \ln \epsilon^{-1}$. There could also be tension between Goals 1 and 3: even in the offline setting, it is conceivable that, to achieve $d = \ln \epsilon^{-1} + O(1)$, one must place each element x in a roughly random position out of $h_1(x), \ldots, h_d(x)$. This, in turn, would force an expected positive query time of $\Omega(d)$.

This Paper: bubble-up cuckoo hashing. In this paper, we present **bubble-up cuckoo hashing**, an implementation of d-ary cuckoo hashing that achieves all three of the above goals simultaneously:

Theorem 1.1. Let $\alpha \in (0,1)$ be a positive constant, and let $d_{\text{core}} \in O(1)$ be sufficiently large as a function of α. Then, for any $\epsilon \le e^{-d_{\text{core}}}$ satisfying $\epsilon^{-1} \le n^{1/4}$, there exists an implementation of d-ary cuckoo hashing that:

- uses $d = \lceil \ln \epsilon^{-1} + \alpha \rceil$;

- achieves expected insertion time $O(\delta^{-1})$ for any insertion taking place at a load factor $1 - \delta \le 1 - \epsilon$;

- achieves expected positive query time $O(1)$;

- can support $(1 - \epsilon)n$ insertions with a total failure probability $n^{-\Omega(1)}$.

To parse Theorem 1.1's bounds, it is helpful to consider what happens if we set α to be, say, 0.1. In this case, the first two bullets of the theorem say that, for all sufficiently small ϵ, it is possible to support load factor up to $1 - \epsilon$ using $d = \lceil \ln \epsilon^{-1} + 0.1 \rceil$, and while supporting efficient insertions. For perspective, even *offline* solutions require $d \ge \lceil \ln \epsilon^{-1} \rceil$. Thus, if $\lceil \ln \epsilon^{-1} + 0.1 \rceil = \lceil \ln \epsilon^{-1} \rceil$, then Theorem 1.1 achieves the *exact optimal offline d*, while supporting efficient insertions.

The third bullet point of Theorem 1.1, on the other hand, bounds query time: it says that, for any element x that is in the hash table, the expected number of probes needed for a query to find it is $O(1)$. Critically, this $O(1)$ time bound holds even if d and ϵ^{-1} are large – that is, it treats all three of d, ϵ^{-1}, n as asymptotic parameters.

In specifying an implementation of d-ary cuckoo hashing, there are two algorithmic knobs that we can play with. The first algorithmic knob is the ***insertion policy***: when we need to place an item x in one of its positions $h_1(x), \ldots, h_d(x)$, the insertion policy chooses which position to use. This choice is especially important in the (common) case where all d positions are already occupied, in which case what we are really choosing is which of d elements we are going to "evict". The evicted element will, in turn, need to pick from one of its d choices, and so on. The second algorithmic knob is the ***query policy***: when we query an element x, the query policy decides in what order we should examine the positions $\{h_i(x)\}$? This may seem like a silly distinction at first glance (why not just use the order $i = 1, 2, 3, \ldots$?), but it will turn out to be surprisingly important for bounding the expected time for positive queries (the third bullet point of Theorem 1.1).

The reason that we call our algorithm *bubble-up cuckoo hashing* is because of how it implements insertions. For any given element x, the choice $\texttt{choice}(x)$ for which hash function $h_{\texttt{choice}(x)}$ it uses has a tendency to "bubble up" over time. At any given moment, there is some value $d_{\max}$ dictating the maximum value of $\texttt{choice}(x)$ over all elements x in the table. When an element x is evicted, it prefers to increase its value of $\texttt{choice}(x)$. Only when an element reaches $\texttt{choice}(x) = d_{\max} - O(1)$ does the element become willing to have $\texttt{choice}(x)$ decrease; and even then, it keeps $\texttt{choice}(x)$ within $O(1)$ of $d_{\max}$. Over time, the parameter $d_{\max}$ also increases, so that elements x that were "at the surface" (i.e., $\texttt{choice}(x)$ was $d_{\max} - O(1)$) can once again continue bubbling up.

The quantity $d_{\max}$ ends up also being important for our query policy: rather than examining the hashes in the order of $h_1(x), h_2(x), \ldots$, we examine them in the order of $h_{d_{\max}}(x), h_{d_{\max}-1}(x), h_{d_{\max}-2}(x), \ldots$. Intuitively, because elements x tend to "bubble up" in their value of $\texttt{choice}(x)$ over time, most elements x will be in a position of the form $h_{d_{\max}-j}(x)$ for some relatively small j. In fact, for any element that is in the hash table, we will show that the query time is bounded by a geometric random variable with mean $O(1)$.

To present bubble-up cuckoo hashing as cleanly as possible, we will begin in Section 4 with a warmup version of the algorithm which we call ***basic bubble-up cuckoo hashing***. Even this basic version of the algorithm achieves a nontrivial guarantee: supporting $O(\delta^{-1})$-time insertions with $d = O(\ln \epsilon^{-1})$.

The advantage of starting with basic bubble-up cuckoo hashing is that it is *remarkably simple*. Both the algorithm and its analysis could reasonably be taught in a data structures course.

After presenting the basic version of the algorithm, we continue to Section 5 to present **advanced bubble-up cuckoo hashing**. This is the algorithm achieves the full set of guarantees in Theorem 1.1. The analysis of the algorithm ends up requiring quite a few more ideas than its simpler counterpart, but nonetheless is able to build on some of the same basic principles.

2   Related Work

The idea of using $d > 2$ hash functions was first proposed by Fotakis, Pagh, Sanders, and Spirakis [13] in 2005. The authors showed that, using the breadth-first-search insertion policy, it is possible to set $d = \Theta(\log \epsilon^{-1})$ while supporting insertions in expected time $\epsilon^{-O(\log \log \epsilon^{-1})}$.

Even in the offline setting, it is interesting to study the **maximum load threshold** c_d^* that a d-ary cuckoo hash table can support. This threshold c_d^* is the maximum value such that, as n goes to infinity, it becomes possible to support any load factor of the form $(1 - \Omega(1))c_d^*$. Several independent works [8, 15, 18] have characterized this load threshold, showing, in particular, that is equivalent to the previously known thresholds for the random k-XORSAT problem [8]. One consequence is that, as $d \to \infty$, the threshold c_d^* behaves as $1 - e^{-d} - e^{-o(d)}$.

In the online setting, there has been a great deal of interest in analyzing the *random-walk* insertion policy. Frieze, Melsted, and Mitzenmacher [19] study the *worst-case* insertion time of this strategy. They show that, for any load factor $1 - \epsilon \in (0, 1)$, there exists $d = \Theta(\log \epsilon^{-1})$ such that the worst-case insertion time is polylogarithmic with probability $1 - o(1)$. This result was subsequently tightened by Fountoulakis, Panagiotou, and Steger [16] to support load factors closer to c_d^*.

The expected insertion time of random-walk insertions has proven quite tricky to analyze. Frieze and Johansson [17] show that, for any constant $\epsilon \in (0, 1)$, there exists a d_ϵ such that for all $d > d_\epsilon$, such that random-walk insertions at a load factor of $1 - \epsilon$ support $O(1)$ expected insertion time. Very recently, Bell and Frieze [3] have proven a stronger result: they show that, so long as $4 \le d \le O(1)$, and so long as the load factor is at most $(1 - \Omega(1))c_d^*$, then the expected insertion time is $O(1)$. The state of the art for $d = 3$ remains a result by Walzer [31] who proves $O(1)$ expected insertion time for load factors up to 0.818.

Eppstein, Goodrich, Mitzenmacher, and Pszona [10] consider the task of minimizing the *wear* of the hash table, defined to be the *maximum number of times that any single item gets moved*. They show that, for $d = 3$, there exists an insertion algorithm that fills the hash table to load factor $\Omega(1)$ while guaranteeing a maximum wear of at most $\log \log n + O(1)$ with high probability.

Khosla and Anand [22] consider the task of proving a *high probability* bound on the *total insertion time* needed to fill a d-ary cuckoo hash table to load factor $1 - \epsilon$. They show that, if $d = O(1)$ and if $1 - \epsilon = (1 - \Omega(1))c_d^*$, then there is an insertion algorithm that achieves a high-probability insertion-time bound of $O(n)$.

An alternative to d-ary cuckoo hashing is *bucketized cuckoo hashing*, introduced by Dietzfelbinger and Weidling [9] in 2007. In this setting, each key hashes to two *buckets* of size $b > 1$. Dietzfelbinger and Weidling [9] studied the breadth-first-search insertion policy, and showed that, for $b \ge 16 \ln \epsilon^{-1}$, the policy achieves $O(\epsilon^{-\log \log \epsilon^{-1}})$ expected-time insertions. They left as an open question whether one could prove a similar result for random-walk insertions. The closest such result is due to Frieze and Petti [20], who prove the following: if $b \ge \Omega(\epsilon^{-2} \log \epsilon^{-1})$, and if insertions are implemented using random-walk evictions, then the hash table can be filled to a load factor up to $1 - \epsilon$ while supporting insertions with eviction chains of expected length $O(1)$. Bucketized cuckoo hashing has also been studied in the offline setting, where the goal is to determine the critical load threshold for any given number d of buckets and any given bucket size b; here, results are known both for non-overlapping [12, 6, 14, 25] and overlapping [24, 32] buckets.

3

Finally, another generalization of d-ary cuckoo hashing is the setting in which the d hashes need not be independent. Here, a common technique is the use of a *backyard*: items hash to bins of some size d_1, and if the bin is overloaded, then the item is placed into a (much smaller) secondary data structure known as the backyard [2, 21, 5, 30, 4]. If the backyard itself is a cuckoo hash table (or a deamortized cuckoo hash table [1]), then the resulting data structure is known as a *backyard cuckoo hash table* [2]. Backyard cuckoo hashing can be used to support a load factor of form $1 - \epsilon$ with $d = \tilde{O}(\epsilon^{-2})$ [2, 4]. Although this value of d is exponentially larger than the target value of $d = \log \epsilon^{-1} + O(1)$ in the current paper, backyard cuckoo hashing turns out to be nonetheless a quite useful algorithmic primitive in the design of constant-time succinct hash tables [2].

3   Preliminaries

Cuckoo hashing terminology. A d-ary cuckoo hash table stores elements in an array of size n, where each element x is guaranteed to be in one of positions $h_1(x), h_2(x), \ldots, h_d(x)$. As in past work [8, 15, 18, 13, 19, 16, 17, 3, 31, 10, 22], will assume that the hash functions $h_1, h_2, \ldots, h_d$ are each fully independent and uniformly random.

The **load factor** of a hash table is the fraction m/n, where m is the current number of elements. We will typically use $1 - \epsilon$ to denote the maximum load factor that the hash table supports, and $1 - \delta$ to denote the current load factor. Our goal will be to support insertions in time $O(\delta^{-1})$.

When an element x is inserted, the insertion will perform an **eviction chain** in which x is placed in some position j_1, the element x_2 formerly in position j_1 is placed in some position j_2, and so on, with the final element in the chain being moved into a free slot. Each of the elements $x_2, x_3, \ldots$ are said to have been **evicted** during the insertion. Insertions are permitted to declare **failure**. In practice, one can rebuild the hash table whenever a failure occurs—so long as the overall failure probability (for all insertions) is $o(1)$, the cost of these rebuilds is a low-order term in the overall cost of the insertions.

For a given element x and index $i \in [d]$, we say that we have **first-time probed** $h_i(x)$ if, during some insertion, we have evaluated $h_i(x)$ and checked whether that slot was empty.

For a given element x in the hash table, let $\texttt{choice}(x)$ denote the index i of the hash function h_i that x is currently using. If x is not in the hash table, $\texttt{choice}(x) := 0$. To simplify our discussion, we will assume throughout the body of the paper that $\texttt{choice}$ can be evaluated in constant time. We will then show in Appendix B how to replace $\texttt{choice}$ with an explicit protocol $\texttt{choice}'$ that (with a bit of additional algorithmic case handling) preserves both the time and correctness bounds established in the body of the paper.

Background machinery. Our warmup algorithm in Section 4 will make use of the classic 2-ary cuckoo hashing analysis from [29]. Note that, in 2-ary cuckoo hashing, there is only one possible eviction chain that an insertion can follow, so there is no algorithmic flexibility in the insertion strategy.

Theorem 3.1 (2-ary cuckoo hashing [29]). Consider a 2-ary cuckoo hash table where we declare failure on an insertion if it takes time $\omega(\log n)$. Then, for any sequence of $n/2 - \Omega(n)$ insertions, we have that:

- The probability of any insertion failing is $O(1/n)$.

- The expected time per insertion is $O(1)$.

We remark that the use of $\omega(\log n)$ in Theorem 3.1 is a slight abuse of notation: What we mean by "declaring failure if an insertion takes time $\omega(\log n)$" is that the insertion algorithm should select some sufficiently large positive constant c and declare failure after time $c \log n$. The use of ω notation

4

here, and throughout the paper when specifying failure conditions, is just so that we do not have to carry around extra constants needlessly.

The advanced version of our algorithm will require the use of slightly more heavy machinery. In particular, we will apply a recent result by Bell and Frieze [3] who analyze the d-ary **random-walk** eviction strategy in which, when an element x is evicted, it selects a random $i \in [d]$ and goes to position $h_i(x)$. The authors show that, at load factors of the form $1 - \Omega(1)$, random-walk insertions take $O(1)$ expected time:

Theorem 3.2 (Random-walk d-ary cuckoo hashing [3]). There exists functions $D : \mathbb{N} \to \mathbb{R}^+$, $T : \mathbb{N} \to \mathbb{R}^+$, $N : \mathbb{N} \to \mathbb{R}^+$, and $F : \mathbb{N} \to \mathbb{R}^+$ satisfying $\lim_{d \to \infty} D(d) = 0$, and such that the following is true. Let $d \in \mathbb{N}$, and consider random-walk d-ary cuckoo hashing on $n > N(d)$ slots, where we declare failure on an insertion if it takes time $T(d) \log^{\omega(1)} n$. Consider any sequence of $n \cdot (1 - e^{-d+D(d)})$ insertions. Then:

- The probability of any insertion failing is $O(n^{-F(d)})$.

- The expected time per insertion is $O(T(d))$.

As before, we are abusing notation in our description of the termination condition: terminating after $T(d) \log^{\omega(1)} n$ steps just means that we terminate after $T(d) \log^c n$ steps for some sufficiently large positive constant c. It is also worth remarking that we will be applying Theorem 3.2 only to the case of $d = O(1)$, so the failure probability $O(n^{-F(d)})$ will be $n^{-\Omega(1)}$, and the time $O(T(d))$ will be $O(1)$.

Finally, we will also need some machinery for proving concentration bounds. Specifically, we will make use of McDiarmid's inequality [27], which can also be viewed as Azuma's inequality applied to a Doob martingale.

Theorem 3.3 (McDiarmid's Inequality [27]). Call a function $f(x_1, x_2, \ldots, x_n) : U^n \to \mathbb{R}$ C-Lipschitz if changing the value of a single x_i can only ever change f by at most C. Given a C-Lipschitz function f, and given independent random variables $X_1, X_2, \ldots, X_n \in U$, the random variable $F = f(X_1, \ldots, X_n)$ satisfies

$$\Pr[|F - \mathbb{E}[F]| \geq jC\sqrt{n}] \leq e^{-\Omega(j^2)}$$

for all $j > 0$.

Other general notation. We say that an even occurs with **high probability in** n, or equivalently that it occurs with probability $1 - 1/\operatorname{poly}(n)$, if, for all positive constants c, the event occurs with probability at least $1 - O(1/n^c)$. We will use $[a, b]$ to denote $\{a, a+1, \ldots, b\}$ and $(a, b]$ to denote $\{a+1, a+2, \ldots, b\}$. Finally, when using asymptotic notation, we will always treat all of $n, d, \epsilon^{-1}, \delta^{-1}$ as asymptotic variables.

A remark on deletions. Although our focus is on insertions and queries, it is worth noting that one can also add deletions in a black-box fashion. Indeed, using *tombstones* to implement deletions results in the following corollary of Theorem 1.1, proven in Appendix C.

Corollary 3.1. Let $\alpha \in (0, 1)$ be a positive constant, and let $d_{\text{core}} \in O(1)$ be sufficiently large as a function of α. Then, for any $\epsilon \leq e^{-d_{\text{core}}}$ satisfying $\epsilon^{-1} \leq n^{1/4}$, there exists an implementation of d-ary cuckoo hashing that supports both insertions and deletions and that:

- uses $d = \lceil \ln \epsilon^{-1} + \alpha \rceil$;

- achieves amortized expected time $O(\delta^{-1} \log \delta^{-1})$ for any insertion/deletion taking place at a load factor $1 - \delta \leq 1 - \epsilon$;

- achieves expected positive query time $O(1)$.

4   Warmup: The Basic Bubble-Up Algorithm

We begin by presenting a basic version of the bubble-up algorithm that achieves the following simple guarantee:

Theorem 4.1. Let $\epsilon \in (n^{-1/4}, 1)$, and let $d = \lceil 3 \ln \epsilon^{-1} \rceil + 1$. The basic bubble-up algorithm is a d-ary eviction policy that supports $(1 - \epsilon)n$ insertions with probability $1 - O(1/n)$, and where the expected time for the $(1 - \delta)n$-th insertion is $O(\delta^{-1} + \log \epsilon^{-1})$, for any $\delta \geq \epsilon$.

The basic bubble-up algorithm.   The basic bubble-up algorithm uses the following strategy for inserting/evicting an element x:

- If $\texttt{choice}(x) = d$, move x to $h_{d-1}(x)$, and evict anyone who is there. **(Type 1 Move)**

- If $\texttt{choice}(x) = d - 1$, move x to $h_d(x)$, and evict anyone who is there. **(Type 2 Move)**

- If $\texttt{choice}(x) < d - 1$, sequentially check if any of $h_{\texttt{choice}(x)+1}(x), \ldots, h_{d-2}(x)$ are free slots. If any of them are free slots, use the first one found **(Type 3 Move)**; otherwise, move x to $h_{d-1}(x)$ and evict anyone who is there **(Type 4 Move)**.

- Finally, if we make $\omega(\log n)$ Type 1 or Type 2 moves in a row, without finding a free slot, declare **failure**.

Analysis.   At a high-level, one should think of there as being two types of elements: those using their first $d - 2$ hashes (**non-core elements**); and those using their final 2 hashes (**core elements**). The non-core elements interact passively with the table: a non-core element would like to use one of its first $d-2$ hashes, if it can, but is not willing to evict another element to do so. On the other hand, once an element "bubbles up" to become a core element, it interacts more actively with the table: whenever a core element x is evicted, it goes to whichever position $h_d(x), h_{d-1}(x)$ it is not currently in, and it evicts any element that is there. Morally, one should think of the core elements as forming a 2-ary cuckoo hash table (the **core hash table**) that *lives in the same slots* as the full d-ary cuckoo hash table. The elements in the core hash table take priority over those that are not, meaning that core elements can evict non-core elements, but not vice-versa.

The main step of the analysis will be bounding the number of elements that get placed into the core hash table. What we will see is that most elements are able to make use of their first $d - 2$ hashes, and that only a small fraction make it into the core hash table. This, in turn, is what allows the core hash table to function correctly, despite the fact that it is only a 2-ary cuckoo hash table.

The analysis, and indeed all of the analyses in this paper, will make critical use of the following coupon-collector identity, proven in Appendix A.

Proposition 4.1. Let $\epsilon \in (n^{-1/4}, 1)$. Suppose we sample iid uniformly random coupons $u_1, u_2, \ldots \in [n]$, stopping once we have sampled a total of $(1 - \epsilon)n$ distinct coupons. With probability $1 - 1/\operatorname{poly}(n)$, the number of sampled coupons is

$$n \ln \epsilon^{-1} \pm \tilde{O}(n^{-1/4}).$$

We can apply Proposition 4.1 to our setting as follows. Each first-time probe can be viewed as a coupon, sampling a uniformly random slot in $[n]$. The $(1 - \epsilon)n$ insertions terminate once we have sampled $(1 - \epsilon)n$ distinct slots. Thus, Proposition 4.1 tells us that, with probability $1 - 1/\operatorname{poly}(n)$, either the eviction policy fails, or the total number of first-time probes made is $n \ln \epsilon^{-1} \pm \tilde{O}(n^{-1/4})$.

Recall that we call an element x a **core** element if $\texttt{choice}(x) \in \{d - 1, d\}$. We can use Proposition 4.1 to derive a high-probability bound on the number of core elements:

Lemma 4.2. With probability $1 - 1/\operatorname{poly}(n)$, the number of core elements is at most $(1 + o(1))n/3$.

Proof. Let P be the total number of first-time probes that we perform, and let K be the total number of core elements after all the insertions are complete. If an element x is a core element, then we must have first-time probed all of $h_1(x), \ldots, h_{d-1}(x)$. It follows that

$$P \ge (d-1)K \ge 3K \ln \epsilon^{-1}.$$

On the other hand, by Proposition 4.1, we have with probability $1 - 1/\operatorname{poly}(n)$ that

$$P \le (1 + o(1))n \ln \epsilon^{-1}.$$

Chaining together these identities, we get that

$$3K \ln \epsilon^{-1} \le (1 + o(1))n \ln \epsilon^{-1}.$$

which implies that $K \le (1 + o(1))n/3$, as desired. $\qquad\square$

An important feature of core elements is that, whether or not an element x is core has nothing to do with $h_{d-1}(x)$ or $h_d(x)$. We can formalize this in the following lemma, which we refer to as the *core independence property*.

Lemma 4.3 (The Core Independence Property). Let x be an element, and let C be the event that x is core. The event C is independent of the pair $(h_{d-1}(x), h_d(x))$.

Proof. This follows from two observations: (1) Once an element x becomes a core element, it stays a core element; and (2) Prior to an element x becoming a core element, we never evaluate $h_{d-1}(x)$ or $h_d(x)$. $\quad\square$

As noted earlier, we can think of the core elements as forming a **core hash table** in which the only hash functions are h_{d-1} and h_d. An element x is inserted into the core hash table when it becomes a core element. The insertion triggers a sequence of Type 1 and Type 2 moves, that correspond to an eviction chain in the core hash table. The eviction chain ends once it encounters a slot that does not contains a core element (i.e., a slot that the core hash table thinks of as empty). Thus, the core hash table operates exactly like a standard 2-ary cuckoo hash table.

Lemma 4.2 tells us that, with high probability, the core hash table never has more than $n/3 - o(n) \le n/2 - \Omega(n)$ elements. Lemma 4.3, on the other hand, tells us that the hash functions h_{d-1} and h_d used within the core hash table are fully random. Combined, the lemmas allow us to apply the classical analysis of 2-ary cuckoo hashing (Theorem 3.1) to deduce that:

- **Fact 1:** Each eviction chain in the core hash table has expected length $O(1)$;

- **Fact 2:** The probability that any eviction chain in the core hash table ever has length $\omega(\log n)$ (this includes the failure event where the insertion fails) is at most $O(1/n)$.

Recall that the basic bubble-up algorithm fails if there are ever $\omega(\log n)$ Type 1 and Type 2 steps in a row. Fact 2 tells us that the probability of such a failure ever occurring is $O(1/n)$.

Fact 1, on the other hand, can be used to bound the expected insertion time, overall, within the (full) hash table. Consider the $((1 - \delta)n + 1)$-th insertion. Let Q be the number of first-time probes made by the insertion. Since each first-time probe has at least a δ probability of finding a free slot, we have that

$$\mathbb{E}[Q] = O(\delta^{-1}).$$

On the other hand, we can bound the total time T spent on the insertion by the sum of two terms:

- T_1 is the time spent on Type 1 and Type 2 moves;

- T_2 is the number of first-time probes made by Type 3 and Type 4 moves.

By construction $T_2 \leq O(Q)$, so $\mathbb{E}[T_2] = O(\delta^{-1})$. To bound T_1, define J to be the total number of core-table eviction chains that occur during the current insertion. By Fact 1, we have that

$$\mathbb{E}[T_1] \leq O(J).$$

On the other hand, each core-table eviction chain is triggered by an element x becoming a core element, at which point $h_{d-1}(x)$ experiences a first-time probe. It follows that $J \leq Q$, which implies that $\mathbb{E}[T_1] \leq O(\delta^{-1})$. Thus $\mathbb{E}[T_1 + T_2] = O(\delta^{-1})$, as desired.

5 The Advanced Bubble-Up Algorithm

In this section, we present the advanced bubble-up algorithm, which achieves the main result of the paper:

Theorem 1.1. Let $\alpha \in (0, 1)$ be a positive constant, and let $d_{\text{core}} \in O(1)$ be sufficiently large as a function of α. Then, for any $\epsilon \leq e^{-d_{\text{core}}}$ satisfying $\epsilon^{-1} \leq n^{1/4}$, there exists an implementation of d-ary cuckoo hashing that:

- uses $d = \lceil \ln \epsilon^{-1} + \alpha \rceil$;

- achieves expected insertion time $O(\delta^{-1})$ for any insertion taking place at a load factor $1 - \delta \leq 1 - \epsilon$;

- achieves expected positive query time $O(1)$;

- can support $(1 - \epsilon)n$ insertions with a total failure probability $n^{-\Omega(1)}$.

5.1 Technical Overview: Motivating the Advanced Bubble-Up Algorithm.

To motivate the proof of Theorem 1.1, let us first revisit the basic bubble-up algorithm from Section 4, which used $d = 3 \ln \epsilon^{-1} + O(1)$. There are two bottlenecks preventing us from reducing d.

First bottleneck: the core table. The first bottleneck is that, since the core hash table is 2-ary, it requires a load factor smaller than $1/2$. This bottleneck is easy to rectify: Just implement the core hash table as a d_{core}-ary hash table for some large constant d_{core}. This allows us to store up to, say, $0.99n$ elements in the core hash table, while still ensuring that it operates efficiently [3]. With this modification to the basic bubble-up algorithm, one can improve the bound on d to, say, $d = 1.02 \ln \epsilon^{-1} + O(1)$. Of course, although this is an improvement over $3 \ln \epsilon^{-1} + O(1)$, it is still a far cry from the bound of $\lceil \ln \epsilon^{-1} + \alpha \rceil$ that we will ultimately want – this is where the second bottleneck comes into play.

Second bottleneck: the number of first-time probes by elements not in the core table. The second bottleneck is more significant. In the analysis of the basic bubble-up algorithm, we can argue that each element x in the core hash table has made at least $d - d_{\text{core}} + 1$ first-time probes. But we cannot say anything about the elements x *not* in the core hash table. This is not just a problem with the analysis – the elements not in the core hash table really are likely to make much fewer than d first-time probes.

To see why this is a problem, recall that, overall, we need to achieve roughly $n \ln \epsilon^{-1}$ first time probes (Proposition 4.1). Now imagine for a moment that both of the following facts are true: (1) we are using d only slightly larger than $\ln \epsilon^{-1}$; and (2) the elements *not* in the core hash table make, on average, much fewer than d total probes. Then, the only way for the total number of first-time probes to be roughly

$n \ln \epsilon^{-1}$ is if the vast majority of the elements are in the core hash table! This would be a disaster for the core hash table, as it is only capable of supporting load factors of the form $1 - \Omega(1)$.

Thus, if we want to get away with a small value of d, in particular, a value of the form $\ln \epsilon^{-1} + O(1)$, we need to ensure that almost all of the elements, *including those not in the core table*, end up achieving *very close* to d probes.

To achieve this guarantee, we introduce a critical modification to the algorithm: We set a parameter $d_{\max}$ that increases over time, and where, at any given moment, all of the elements use only their first $d_{\max}$ hashes. At a high level, the rule for increasing $d_{\max}$ is that, whenever the load factor reaches $1 - e^{-d_{\max}+\alpha}$, for the current value of $d_{\max}$, we increase the value of $d_{\max}$ by d_{core}. We refer to the maximal time window during which $d_{\max}$ takes a given value as a **phase**.

At any given moment, we define the "core hash table" to consist of the elements that are using hash functions $\{h_{d_{\max}-d_{\text{core}}+1}, \ldots, d_{\max}\}$. This means that, whenever a new phase begins, the core hash table *becomes empty*. In order for an element to get added to the core hash table during a given phase, it must be part of an eviction chain that occurs *during that phase*.

Intuitively, this results in the following situation. During a given phase i, a large fraction of elements, say a 0.99 fraction, make it into the core table. These elements will have probed almost all of their hashes. (In fact, by applying Proposition 4.1 to the core hash table, we will be able to argue that the core elements have probed not just their first $d_{\max} - d_{\text{core}}$ hashes, but also, on average, most of their final d_{core} hashes.) However, even among the elements that *do not* make it into the core table in phase i, most of them will have made it into the core table in phase $i - 1$. These elements will *also* have probed all but $O(1)$ of their hashes. Among the elements that do not make it into the core table in phases $i - 1$ or i, most will have made it into the core table in phase $i - 2$, and so on. The result is that most elements will have probed almost all of their hashes. This is how we are able to obtain the first-time probes required by Proposition 4.1 without overflowing the core table, and while using a value of d of the form $\lceil \ln \epsilon^{-1} + \alpha \rceil$.

Analyzing the core table. Of course, the above intuition doesn't actually answer the question of *how* we bound the load of the core hash table. This is the main technical contribution of the proof. The basic idea is to examine four quantities:

- P_1: The total number of first-time probes made prior to phase i.

- P_2: The total number of first-time probes made by the core table during phase i.

- P_3: The total number of first-time probes made outside of the core table (i.e., to $h_i(x)$ for some $i < d_{\max} - d_{\text{core}}$) during phase i.

- P_4: The total number of first-time probes made by the end of phase i.

The quantities P_1 and P_4 are controlled by Proposition 4.1, and together force $P_2 + P_3 = P_4 - P_1$ to be roughly $nd_{\max}$. The second quantity P_2 is also controlled by Proposition 4.1, and is roughly $n \ln \rho^{-1}$, where $1 - \rho$ is the final load factor of the core hash table. This gives us a relationship $n \ln \rho^{-1} + P_3 = nd_{\max} \pm o(n)$ that we can use to relate ρ^{-1} to P_3. To prove an upper bound ρ^{-1}, it suffices to actually prove a lower bound on P_3.

We will argue that P_3 is controlled by a relatively simple random process that can be analyzed with the help of McDiarmid's Inequality [27]. So long as at least $\Omega(n)$ elements make it into the core hash table during the current phase, we will get a high-probability $\Omega(n)$ lower bound on P_3. By balancing various constants appropriately, this will allow us to obtain Theorem 1.1.

What about queries? Finally, the structure of the advanced bubble-up algorithm also comes with a second advantage: positive queries can be completed in *expected* time $O(1)$, no matter how large the

parameter $1 - \epsilon$ may be. This is because, for a given element x, we have with probability at least, say, 0.9, that x uses one of hashes $h_d, \ldots, h_{d-d_{\text{core}}+1}$; and if x doesn't use any of those, then with probability at least 0.9, it uses one of hashes $h_{d-d_{\text{core}}}, \ldots, h_{d-2d_{\text{core}}+1}$, and so on. The result is that, if we examine positions $h_{d_{\max}}(x), h_{d_{\max}-1}(x), \ldots$ one after another, then the time to find x is bounded above by a geometric random variable with mean $O(d_{\text{core}}) = O(1)$.

5.2   The Advanced Bubble-Up Algorithm

In addition to the parameter $\alpha = \Theta(1)$, given to us by Theorem 1.1, the advanced bubble-up algorithm will make use of two parameters $d_{\text{core}} = \Theta(1)$ and $\epsilon_{\text{core}} = \Theta(1)$ that are determined by α. We will first describe the algorithm using these parameters, and then describe how to select the parameters.

Let $\gamma = \lceil \ln \epsilon^{-1} + \alpha \rceil \pmod{d_{\text{core}}}$. We proceed in phases, starting with phase 1, as follows. During a given phase q, define $d_{\max} := \gamma + q d_{\text{core}}$ (implicitly, $d_{\max}$ is a function of q). The phase ends when we reach load factor $1 - e^{-d_{\max}+\alpha}$. The final phase is the one where $d_{\max} = \lceil \ln \epsilon^{-1} + \alpha \rceil$, at the end of which the load factor is at $1 - e^{-\lceil \ln \epsilon^{-1} + \alpha \rceil} \geq 1 - \epsilon$. During a given phase q, we will use only the first $d_{\max}$ hashes of each element.

At any given moment, call an element x a **core** element if $\texttt{choice}(x) \in (d_{\max} - d_{\text{core}}, d_{\max}]$. Our policy for inserting/evicting an element x is:

- **Type 1 Move:** If x is a core element, place it in $h_{d_{\max}-k+1}(x)$ for a random $k \in \{1, \ldots, d_{\text{core}}\}$. If there is an element in that position, it gets evicted.

- **Type 2 Move:** If x is not a core element, then sequentially examine positions $h_i(x)$ for $\max(0, \texttt{choice}(x) - d_{\max}) \leq i \leq d_{\max} - d_{\text{core}}$. If we find a free slot, place x there. Otherwise, x becomes a core element, and we use the procedure for placing a core element.

- **Failures:** Finally, if there are $\log^{\omega(1)} n$ Type 1 moves in a row, without the insertion completing, then declare **failure**.

Note that, by construction, there are no core elements at the beginning of a given phase. The only way for an element to become core is if we have at some point first-time probed all of $h_1(x), \ldots, h_{d_{\max}-d_{\text{core}}}(x)$.

As a convention, we will refer to the core elements as forming a **core hash table**. The core hash table is itself a d_{core}-ary hash table that implements insertions (i.e., additions of new core elements) using random-walk evictions. The failure condition above can be restated as: the overall hash table fails if the core hash table ever has an eviction chain of length $\log^{\omega(1)} n$ (i.e., polylog n for a sufficiently large polylog).

Selecting Parameters. In selecting our parameters d_{core} and ϵ_{core}, we will make black-box use of Theorem 3.2. Let $D : \mathbb{N} \to \mathbb{R}^+$, $T : \mathbb{N} \to \mathbb{R}^+$, and $N : \mathbb{N} \to \mathbb{R}^+$ be the functions from the theorem statement.

We set d_{core} and $\epsilon_{\text{core}} = e^{-d_{\text{core}}+D(d_{\text{core}})}$ so that, for all $j \geq d_{\text{core}}$, we have

$$j/e^{j-\alpha} \leq \alpha/8, \tag{1}$$

and so that

$$\ln \epsilon_{\text{core}}^{-1} + \alpha/8 > d_{\text{core}} \cdot (1 + \epsilon_{\text{core}}). \tag{2}$$

Note that (1) can be achieved simply by setting d_{core} to be sufficiently large as a function of α. The fact that (2) can be achieved follows from the fact that $D(d_{\text{core}}) \to 0$ as $d_{\text{core}} \to \infty$. Indeed, this means that $\ln \epsilon_{\text{core}}^{-1} - d_{\text{core}} \to 0$ and that $\epsilon_{\text{core}} d_{\text{core}} \to 0$ as $d_{\text{core}} \to \infty$, which together ensure that (2) is possible for d_{core} sufficiently large. Although d_{core} is sufficiently large as a function of α, it is still $O(1)$. Therefore, a d_{core}-ary cuckoo hash table, using random-walk evictions, can support insertions at load factors up

to $1 - \epsilon_{\text{core}}$ while supporting $O(1)$ expected-time insertions and a failure probability (cumulative across all insertions) of $n^{-\Omega(1)}$.

Analysis outline. The proof of Theorem 1.1 is split into three parts. We begin in Subsection 5.3 by proving a bound of $1 - \epsilon_{\text{core}}$ on the load factor of the core hash table – this is the most technical part of the analysis. We then complete the analysis of insertions in Subsection 5.4 and of queries in Subsection 5.5.

5.3   Bounding the Number of Core Elements

In this subsection we prove that:

Proposition 5.1. With high probability in n, the number of core elements at the end of any given phase is at most $(1 - \epsilon_{\text{core}})n$.

Throughout the section, we will focus on a fixed phase q, and use $d_{\max}$ to denote the value of $d_{\max}$ during phase q. We break the analysis into two cases, $q = 1$ and $q > 1$, each of which are handled in their own subsection.

Remark 5.2 (A Trick for Handling Failure Events in the Analysis). Recall that the core hash table can sometimes fail, in particular, when an insertion is either impossible or takes $\omega(\log n)$ time. When this happens, the overall hash table also fails. So that we can ignore such failure events in this subsection, it is convenient to define the following revised version of the data structure: Suppose that, whenever an insertion of an element x in the core hash table fails, we resample the core hashes $h_{d_{\max}-d_{\text{core}}+1}(x), \ldots, h_{d_{\max}}(x)$ and we try the insertion again. Similarly, if that insertion fails, we again resample the core hashes, and so on. Note that, if a hash has been resampled, then the first time that it is probed (after being resampled), we consider that probe to, once again, be a first-time probe.

This revised data structure has the feature that it never experiences failures. To prove Proposition 5.1 for the un-revised data structure, it suffices to prove it instead for the revised one. Thus, for the rest of the subsection, we will implicitly consider the revised hash table rather than the un-revised one.

5.3.1   Analyzing phase $q = 1$

Lemma 5.3. During phase $q = 1$, the total number of elements that make it into the core table is, with high probability in n, at most $(1 - \epsilon_{\text{core}})n$.

Proof. We calculate the total number P of first-time probes made during phase 1 in two different ways. At the end of the phase, the load factor is $1 - 1/e^{d_{\text{core}}+\gamma-\alpha}$. Thus, by Proposition 4.1, we have with high probability in n that
$$P = (d_{\text{core}} + \gamma - \alpha)n \pm o(n). \tag{3}$$
On the other hand, we also know that
$$P = P_1 + P_2,$$
where P_1 is the number of first-time probes made in the core table and P_2 is the number of non-core first-time probes. If Q elements make it into the core table, then we have by Proposition 4.1 that, with high probability in n, either $Q \leq (1 - \epsilon_{\text{core}})n$, or
$$P_1 \geq n \ln \epsilon_{\text{core}}^{-1} - o(n).$$
On the other hand, each element that makes it into the core table must first incur γ non-core first-time probes. Therefore, either $Q \leq (1 - \epsilon_{\text{core}})n$, or
$$P_2 \geq \gamma(1 - \epsilon_{\text{core}})n.$$

11

Combining these facts, it follows that, with high probability in n, either $Q \leq (1 - \epsilon_{\text{core}})n$, or

$$P \geq n \ln \epsilon_{\text{core}}^{-1} + \gamma \cdot (1 - \epsilon_{\text{core}})n - o(n). \tag{4}$$

By the construction of ϵ_{core}, and specifically (2), we know that

$$\ln \epsilon_{\text{core}}^{-1} + \alpha/8 > d_{\text{core}} \cdot (1 + \epsilon_{\text{core}}),$$

which, combined with the fact that $\gamma \leq d_{\text{core}}$, implies that

$$\ln \epsilon_{\text{core}}^{-1} + \gamma \cdot (1 - \epsilon_{\text{core}}) > d_{\text{core}} + \gamma - \alpha/8.$$

Therefore, (3) and (4) are contradictory, meaning that (4) happens with probability at most $1/\operatorname{poly}(n)$. Thus, we have with high probability in n that $Q \leq (1 - \epsilon_{\text{core}})n$, as desired. $\qquad\square$

5.3.2 Analyzing a phase $q \geq 2$

For each element x, define $\overline{P}(x)$ to be the total number of first-time probes that are made on x in the first $q - 1$ phases, and define $P(x) = d_{\max} - d_{\text{core}} - \overline{P}(x)$. Intuitively, $P(x)$ is the number of additional first-time probes that x can make, during phase q, before it becomes a core element. A major step in bounding the number of core elements during phase q will be to show that there are many first-time probes performed outside of the core hash table (i.e., first-time probes to the first $d_{\max} - d_{\text{core}}$ hashes). For this, an important technical step is to argue that, on average across the elements x inserted/evicted during the phase, $P(x)$ is non-trivially large:

Lemma 5.4. Suppose $q \geq 2$. Let E be the set of elements that are inserted/evicted during phase q. With high probability in n, we have

$$\sum_{x \in E} P(x) \geq |E| \cdot \alpha/2 - o(n).$$

Proof. Let us break E into E_1, consisting of elements inserted during the phase, and E_2, consisting of elements that are not inserted during the phase but that are evicted at least once. We know that $\sum_{x \in E_1} P(x) = |E_1| \cdot (d_{\max} - d_{\text{core}}) > \alpha |E_1|$, so it suffices to show that

$$\sum_{x \in E_2} P(x) \geq |E_2| \cdot \alpha/2 - o(n).$$

Let V be the set of elements present at the beginning of the phase. Let us consider how E_2 evolves over time, during the phase. A critical observation is that, whenever $|E_2|$ increases, the new element x that is added to E_2 is *uniformly random* out of the elements in V that are not yet in E_2 – this is because the element y that is evicting x found x via a first-time probe, and first-time probes are uniformly random. Thus we can think of E_2 as being generated by: starting with an empty set, and repeatedly adding random elements from V until some stopping condition is met.

Equivalently, for the sake of analysis, we can think of E_2 as being generated by the following fictitious process. Let $v_1, v_2, \ldots$ be independent uniformly random samples of V. Then E_2 is generated by adding $v_1, v_2, \ldots$ to E_2 (where some additions are no-ops since the element has already been added in the past) until some stopping condition is met.

12

Without loss of generality, the sequence $v_1, v_2, \ldots$ has total length $O(n \log n)$, since with high probability in n, such a sequence will hit every element in V. Therefore, if we define $V_t = \{v_1, v_2, \ldots, v_t\}$, we have

$$|E_2| \cdot \alpha/2 - \sum_{x \in E_2} P(x) \leq \max_{t \in O(n \log n)} \left(|V_t| \cdot \alpha/2 - \sum_{x \in V_t} P(x) \right).$$

To bound this quantity with high probability, it suffices to consider a fixed $t \in O(n \log n)$ and to prove that, with high probability in n, we have

$$|V_t| \cdot \alpha/2 - \sum_{x \in V_t} P(x) \leq o(n).$$

Say that the hash table has a ***natural*** state at the beginning of phase q if, conditioned on that state, the quantity $|V_t| \cdot \alpha/2 - \sum_{x \in V_t} P(x)$ has expected value at most $o(n)$.

If we condition on the hash table being in a natural state at the beginning of phase q, then we can analyze $|V_t| \cdot \alpha/2 - \sum_{x \in V_t} P(x)$ as follows. Because $v_1, \ldots, v_t$ are independent random variables, and because changing any given v_i can change $|V_t| \cdot \alpha/2 - \sum_{x \in V_t} P(x)$ by at most $d_{\max} - d_{\mathrm{core}} = O(\log n)$, we can apply McDiarmid's inequality (Theorem 3.3) to conclude that, with high probability in n, $|V_t| \cdot \alpha/2 - \sum_{x \in V_t} P(x)$ deviates from its mean by at most $\tilde{O}(\sqrt{n})$. This, in turn, implies that $|V_t| \cdot \alpha/2 - \sum_{x \in V_t} P(x) \leq o(n) + \tilde{O}(\sqrt{n}) = o(n)$.

Thus, it remains only to show that, with high probability in n, the hash table is, in fact, in a natural state at the beginning of phase q. Note that the hash table is in a natural state if and only if, for a random element $v \in V$,

$$\mathbb{E}[P(v)] \geq \alpha/2 - o(1).$$

This, expands to

$$\mathbb{E}[\overline{P}(v)] \leq d_{\max} - d_{\mathrm{core}} - \alpha/2 + o(1),$$

which is equivalent to saying that the total number of first-time probes made in the first $q - 1$ phases is less than or equal to

$$|V| \cdot (d_{\max} - d_{\mathrm{core}} - \alpha/2 + o(1)).$$

By construction, the load factor at the beginning of phase $q-1$ is $1 - 1/e^{d_{\max} - d_{\mathrm{core}} - \alpha}$. This implies by Proposition 4.1 that the total number of first-time probes in the first $q-1$ phases is (with high probability) at most

$$n \cdot (d_{\max} - d_{\mathrm{core}} - \alpha + o(1)).$$

Therefore, to complete the lemma, it suffices to show that

$$|V| \cdot (d_{\max} - d_{\mathrm{core}} - \alpha/2) \geq n \cdot (d_{\max} - d_{\mathrm{core}} - \alpha).$$

Since $|V| = (1 - 1/e^{d_{\max} - d_{\mathrm{core}} - \alpha})n$, this reduces to showing that

$$n/e^{d_{\max} - d_{\mathrm{core}} - \alpha} \cdot (d_{\max} - d_{\mathrm{core}} - \alpha/2) \leq n\alpha/2.$$

This, in turn, follows from the definition of d_{core}, which by (1), satisfies

$$j/e^{j - \alpha} \leq n\alpha/2$$

for all $j \geq d_{\mathrm{core}}$. $\qquad\square$

13

Building on Lemma 5.4, we can prove a lower bound on the total number P of *non-core* first-time probes made during phase q. The reason that we care about this lower bound is that we will be able to use it (indirectly) to obtain an *upper bound* on the total number of *core* first-time probes made during the same phase.

Lemma 5.5. Suppose $q \geq 2$. Let Q be the number of elements during phase q that move into the core hash table. With high probability in n, either $Q < n/4$, or the number P of non-core first-time probes made during phase q satisfies

$$P \geq Q \cdot \alpha/8.$$

Proof. Let E be the set of elements that are inserted/evicted during phase q. Let E_1 be the subset of E that end up in the core hash table, and E_2 be the subset of E that do not. Then,

$$P \geq \sum_{x \in E_1} P(x) \geq \sum_{x \in E} P(x) - |E_2| \cdot (d_{\max} - d_{\text{core}}) \geq \sum_{x \in E} P(x) - (d_{\max} - d_{\text{core}})n/e^{d_{\max} - d_{\text{core}} - \alpha},$$

where the final step uses the fact that the number of insertions in phase q is less than $n/e^{d_{\max} - d_{\text{core}} - \alpha}$, and that each eviction chain adds at most one element to E_2. By the construction of d_{core}, and specifically by (1), we know that for all $j \geq d_{\text{core}}$, we have $j/e^{j - \alpha} \leq \alpha/8$. Therefore,

$$P \geq \sum_{x \in E} P(x) - \alpha n/8.$$

Finally, applying Lemma 5.4 gives that, with high probability in n, either $Q < n/4$ or

$$P \geq \alpha Q/2 - \alpha n/8 + o(n) \geq \alpha n/4 - \alpha n/8 + o(n) \geq \alpha n/8 + o(n).$$

$\square$

Finally, we can bound the number of elements that make it into the core table during phase q.

Lemma 5.6. During any phase $q \geq 2$, the total number of elements that make it into the core table is, with high probability in n, at most $(1 - \epsilon_{\text{core}})n$.

Proof. We calculate the total number P of first-time probes made during phase q in two different ways. First, by Proposition 4.1, we know that

$$P = d_{\text{core}}n \pm o(n), \tag{5}$$

with high probability in n. On the other hand, we also know that

$$P = P_1 + P_2,$$

where P_1 is the number of first-time probes made in the core table and P_2 is the number of non-core first-time probes. If Q elements make it into the core table, then we have by Proposition 4.1 that, with high probability in n, either $Q \leq (1 - \epsilon_{\text{core}})n$, or

$$P_1 \geq n \ln \epsilon_{\text{core}}^{-1} - o(n).$$

We also have by Lemma 5.5 that, with high probability in n, either $Q \leq (1 - \epsilon_{\text{core}})n$, or

$$P_2 \geq n\alpha/8 - o(n).$$

14

Combining these facts, it follows that, with high probability in n, either $Q \leq (1 - \epsilon_{\text{core}})n$, or

$$P \geq n \ln \epsilon_{\text{core}}^{-1} + n\alpha/8 - o(n). \tag{6}$$

By the construction of ϵ_{core}, and specifically by (2), we know that $\ln \epsilon_{\text{core}}^{-1} + \alpha/8 > d_{\text{core}} + \Omega(1)$. Therefore, (5) and (6) are contradictory, meaning that (6) happens with probability at most $1/\operatorname{poly}(n)$. Thus, we have with high probability in n that $Q \leq (1 - \epsilon_{\text{core}})n$, as desired. $\qquad\square$

5.4 Bounding insertion time and failure probability

We can now bound the insertion time and failure probability for the advanced bubble-up algorithm. This part of the analysis follows a very similar path to the one used for the basic bubble-up algorithm, except that now we use Proposition 5.1 in place of Lemma 4.2 and Theorem 3.2 in place of Theorem 3.1.

We already know from Proposition 5.1 that, with high probability in n, the number of core elements at any given moment is at most $(1 - \epsilon_{\text{core}})n$. In addition to this, we will need what we call the *core independence property*:

Lemma 5.7 (The Core Independence Property). For each element x, whether x gets placed in the core hash table during a given phase is independent of the hashes $h_{d_{\max} - d_{\text{core}} + 1}(x), \ldots, h_{d_{\max}}(x)$.

Proof. This property follows from two observations: (1) Once an element becomes a core element, it stays a core element for the rest of the phase; and (2) Prior to an element x becoming a core element during a given phase, we never evaluate any of $h_{d_{\max} - d_{\text{core}} + 1}(x), \ldots, h_{d_{\max}}(x)$. $\qquad\square$

Lemma 5.7 tells us that we can think of the elements in the core hash table as having d_{core} fully random hashes. This lets us think of the core hash table as a standard d_{core}-ary cuckoo hash table that treats non-core elements as free slots. Whenever a new element is inserted into it (via a Type 2 move), a random eviction chain is performed (via Type 1 moves) until a free slot (i.e., a slot that is either genuinely free or contains a non-core element) is found. Since the load factor of the core hash table is at most $1 - \epsilon_{\text{core}}$ (with high probability), since $\epsilon_{\text{core}} = e^{-d_{\text{core}} + D(d_{\text{core}})}$, and since $d_{\text{core}} = O(1)$, we can apply Theorem 3.2 in order to conclude that:

- **Fact 1:** Each eviction chain in the core hash table has expected length $O(T(d_{\text{core}})) = O(1)$;

- **Fact 2:** The probability that any eviction chain in the core hash table ever has length $T(d_{\text{core}}) \log^{\omega(1)} n = \log^{\omega(1)} n$ (this includes the event that an insertion fails) is at most $n^{-\Omega(1)}$.

Recall that the advanced bubble-up algorithm fails if there are ever $\log^{\omega(1)} n$ Type 1 moves in a row. Fact 2 tells us that the probability of such a failure ever occurring during a given phase is $n^{-\Omega(1)}$. Since the number of phases is $d = O(\log n)$, it follows that the probability of any failures ever occurring is $n^{-\Omega(1)}$.

Fact 1, on the other hand, can be used to bound the expected insertion time, overall, within the (full) hash table. Consider the $((1 - \delta)n + 1)$-th insertion. Let Q be the number of first-time probes made by the insertion. Since each first-time probe has at least a δ probability of finding a free slot, we have that

$$\mathbb{E}[Q] = O(\delta^{-1}).$$

On the other hand, we can bound the total time T spent on the insertion by the sum of two terms:

- T_1 is the number of first-time probes made by Type 1 and moves;

- T_2 is the time spent on Type 2 moves.

By construction, $T_2 \leq O(Q)$, so $\mathbb{E}[T_2] = O(\delta^{-1})$. To bound T_1, define J to be the total number of core-table eviction chains that occur during the current insertion. By Fact 1, we have that

$$\mathbb{E}[T_1] \leq O(J).$$

On the other hand, each core-table eviction chain is triggered by an element x becoming a core element, at which point at least one of $h_{d_{\max}-d_{\mathrm{core}}+1}(x), \ldots, h_{d_{\max}}(x)$ experiences a first-time probe. It follows that $J \leq Q$, which implies that $\mathbb{E}[T_1] \leq O(\delta^{-1})$. Thus $\mathbb{E}[T_1 + T_2] = O(\delta^{-1})$, as desired.

5.5 Bounding positive query time

In this section, we prove the following proposition:

Proposition 5.8 (Bounding positive query time by $O(1)$). Consider an element x that is in the hash table. Then the time to query it by examining $h_{d_{\max}}(x), h_{d_{\max}-1}(x), \ldots$ is bounded above by a geometric random variable with mean $O(1)$.

The basic idea for bounding the query time is as follows. We will show that, within each phase, each element x has probability at least $\Omega(1)$ of being evicted at least once. If x is evicted, then we will argue that it has a good probability of being placed into the core hash table (for the current phase). This means that an $\Omega(1)$ fraction of elements are in the core hash table for the current phase; an $\Omega(1)$ fraction of the *remaining* elements were in the core hash table for the previous phase; and so on. If an element x was in the core hash table j phases ago, then it is currently in a position of the form $h_{d_{\max}-O(j)}(x)$. This means that the element can be queried in time $O(j)$, which we will argue is a geometric random variable with mean $O(1)$. Although this is the basic structure of the proof, the full proof is complicated by two considerations: (1) When an element x is evicted, it does not necessarily get placed into the core hash table; and (2) The hash table can, with some small probability, fail.

We begin by arguing that, whenever an element is inserted/deleted, it is very likely placed into the core table.

Lemma 5.9. Condition on an arbitrary state for the hash table in phase i, and condition on some element x being inserted/evicted at least once during phase i. Then, with probability $1 - O(e^{-i})$, x is placed into the core table for phase i.

Proof. In order for x to not be in the core table, one of the not-yet-probed slots from the sequence $h_1(x), h_2(x), \ldots, h_{d_{\max}-d_{\mathrm{core}}}(x)$ must be free at the beginning of the phase. Since the free-slot density during the phase is $O(e^{-id_{\max}})$, the probability of this occurring is at most

$$(d_{\max} - d_{\mathrm{core}}) \cdot O(e^{-id_{\max}}) = O(d_{\max} e^{-id_{\max}}) = O(e^{-i}).$$

$\square$

Next we bound the probability of certain rare events occurring. Let A_i be the indicator random variable that either phase i fails, or that during phase i, fewer than n first-time probes of the form $h_i(y)$, $i \in (d_{\max} - d_{\mathrm{core}}, d_{\max}]$, are made.

Lemma 5.10. We have $\Pr[A_i = 1] \leq O(1/n)$.

Proof. The probability of phase i failing is $O(1/n)$. Supposing the phase does not fail, by Proposition 4.1, we have with probability $1 - 1/\mathrm{poly}(n)$ that the total number of first-time probes made by the end of phase i is at least $n(d_{\max} - \alpha) - o(n)$. The number of probes made outside of the core table is at most $n(d_{\max} - d_{\mathrm{core}})$, so at least $n(d_{\mathrm{core}} - \alpha) - o(n) > n$ probes are made within the core table, as desired. $\square$

For an element x and a phase i, let $B_{i,x}$ be the indicator random variable event

$$(A_i = 1 \text{ or } x \text{ is inserted/evicted during phase } i),$$

and let $C_{i,x}$ be the indicator random variable event that

$$(A_i = 1 \text{ or } x \text{ gets placed in core hash table during phase } i).$$

Finally, let $B_{<i,x} := \{B_{j,x}\}_{j=1}^{i-1}$ and $C_{<i,x} := \{C_{j,x}\}_{j=1}^{i-1}$.

We will argue that $B_{i,x}$ has probability at least $\Omega(1)$ of being 1, even if we condition on information about earlier phases; and that, if we condition on $B_{i,x}$ being 1, then $C_{i,x}$ is very likely to also be 1.

Lemma 5.11. For an element x that is inserted at some point in the first i phases, and for any outcomes of $B_{<i,x}$ and $C_{<i,x}$, we have

$$\Pr[B_{i,x} = 1 \mid B_{<i,x}, C_{<i,x}] \geq 1 - 1/e$$

and

$$\Pr[C_{i,x} = 1 \mid B_{i,x} = 1, B_{<i,x}, C_{<i,x}] \geq 1 - O(e^{-i}).$$

Proof. If x is inserted during phase i, then $B_{i,x}$ holds trivially. Suppose x is inserted in a previous phase. By definition, either $A_i = 1$ or at least n first-time probes will be made in phase i. For the sake of analysis, if $A_i = 1$, and if the total number of first-time probes that we make is $n' < n$, let us imagine that we make $n - n'$ additional (artificial) first-time probes, so that the total number is n. This means that, regardless of whether $A_i = 1$, we make at least n first-time probes; and that, so long as at least one of these probes finds the position j containing x at the beginning of the phase, then we will have $B_{i,x} = 1$ (in particular, if the probe that finds x is artificial, then this implies $A_i = 1$, which implies $B_{i,x} = 1$ trivially). Thus, to lower bound $\Pr[B_{i,x} = 1 \mid B_{<i,x}, C_{<i,x}]$, it suffices to lower bound the probability that, if we perform n first-time probes, we find the position containing x. This means that

$$\Pr[B_{i,x} = 1 \mid B_{<i,x}, C_{<i,x}] \geq 1 - (1 - 1/n)^n \geq 1 - 1/e,$$

as desired.

Finally, suppose that $B_{i,x}$ occurs, and condition on any outcomes for $B_{<i,x}, C_{<i,x}$. If $B_{i,x}$ occurs without x getting evicted inserted/evicted during phase i, then it must be that A_i occurs, which implies that $C_{i,x}$ occurs. On the other hand, if x gets evicted/inserted during phase i, then the only way for $C_{i,x}$ to be 0 is if, when x gets evicted/inserted, it does not get placed into the core hash table. We know from Lemma 5.9 that the probability of this occurring is at most $O(e^{-i})$, as desired. $\qquad\square$

We now prove that, at the end of any given phase, the query time of each element is bounded above by a geometric random variable. Once we prove this, the only remaining task will be to consider intermediate time points between the ends of phases.

Lemma 5.12. Consider an element x that is inserted at some point in the first i phases. At the end of phase i, let j be the minimum j such that x is in position $h_{d_{\max}-j}(x)$ (and set $j = 0$ if the hash table fails before the end of phase i). Then, j is bounded above by a geometric random variable with mean $O(1)$.

Proof. Note that $j \leq d_{\max} \leq O(i)$ trivially. By Lemma 5.10, we have $\Pr[A_i = 1] = O(1/n) = e^{-\Omega(i)}$. Therefore, $j \cdot A_i$ is bounded above by a geometric random variable with mean $O(1)$. To complete the proof, we will argue that $j \cdot (1 - A_i)$ is also bounded above by a geometric random variable with mean $O(1)$.

Let i_0 be the largest $i_0 \leq i$ such that x was either inserted or evicted at some point during phase i_0. (Note that i_0 itself is a random variable). Observe that, if $C_{i_0,x} = 1$, then either A_i holds, or x is placed into the core hash table during phase i_0. In the latter case, we have $j \leq d_{\text{core}} \cdot (i - i_0 + 1) = O(i - i_0 + 1)$. Thus,

$$j \cdot C_{i_0,x} \cdot (1 - A_i) = O(i - i_0 + 1) \cdot (1 - A_i). \tag{7}$$

On the other hand, if $C_{i_0,x} = 0$, then $j \leq d_{\max} = O(i)$ trivially, so

$$j \cdot (1 - C_{i_0,x}) \cdot (1 - A_i) = O(i). \tag{8}$$

To complete the proof, we will show that each of the left-hand sides of (7) and (8) are bounded above by geometric random variables with means $O(1)$. Indeed, this would imply that $j \cdot (1 - A_i)$ is also bounded above by a geometric random variable with mean $O(1)$, as desired.

For (7), it suffices to show that $(i - i_0) \cdot (1 - A_i)$ is bounded above by a geometric random variable with mean $O(1)$. By the definition of i_0, if $A_i = 0$, then all of $B_{i_0+1,x}, B_{i_0+2,x}, \ldots, B_{i,x}$ are 0. By Lemma 5.11, the probability of this occurring for a given i_0 is at most $(1 - 1/e)^{i-i_0}$. It follows that, for all $k \geq 0$, we have $\Pr[(i - i_0) \cdot (1 - A_i) \geq k] \leq (1 - 1/e)^k$. This means that $(i - i_0) \cdot (1 - A_i)$ is bounded above by a geometric random variable with mean $O(1)$, as desired.

For (8), it suffices to show that $\Pr[C_{i_0,x} = 0 \text{ and } A_i = 0] = e^{-\Omega(i)}$. This would imply that $j \cdot (1 - C_{i_0,x}) \cdot (1 - A_i)$ is non-zero with probability $e^{-\Omega(i)}$. Since, even when $j \cdot (1 - C_{i_0,x}) \cdot (1 - A_i)$ is non-zero, it is at most $O(i)$, it would follow that $j \cdot (1 - C_{i_0,x}) \cdot (1 - A_i)$ is bounded above by a geometric random variable with mean $O(1)$.

By the definition of i_0, we have that $B_{i_0,x} = 1$ and that, if $A_i = 0$, then $B_{i_0+1,x}, \ldots, B_{i,x} = 0$. Thus $\Pr[C_{i_0,x} = 0 \text{ and } A_i = 0]$ is at most the probability that there exists any value k for i_0 such that: (1) $C_{k,x} = 0$, (2) $B_{k,x} = 1$, and (3) $B_{k+1,x}, \ldots, B_{i,x} = 0$. By Lemma 5.11, we have for any given value of k that this occurs with probability at most

$$\Pr[C_{k,x} = 0 \mid B_{k,x} = 1, B_{<k,x}, C_{<k,x}] \cdot \prod_{r=k+1}^{i} \Pr[B_{r,x} = 0 \mid B_{<r,x}, C_{<r,x}] \leq e^{-\Omega(k)} \cdot \prod_{r=k+1}^{i} 1/e \leq e^{-\Omega(i)}.$$

Applying a union bound over the $O(i)$ options for k, the probability of any k existing that satisfies all three of the above conditions is $O(ie^{-\Omega(i)}) = e^{-\Omega(i)}$. This implies that $\Pr[C_{i_0,x} = 0 \text{ and } A_i = 0] = e^{-\Omega(i)}$, as desired. $\qquad\square$

Finally, with a bit of additional casework to handle elements that are inserted during the current (unfinished) phase, we can prove Proposition 5.8.

Proof of Proposition 5.8. If x was inserted prior to the current phase i, then the proposition follows from Lemma 5.12 applied at the end of phase $i - 1$. If x was inserted during phase i, and the hash table does not fail prior to the end of x's insertion, then we have by Lemma 5.11 that, with probability $1 - O(e^{-i})$, x is placed in the core hash table and therefore that $j = O(1)$. In the $O(e^{-i})$-probability case that x is not placed into the core hash table, we have trivially that $j \leq d_{\max} \leq O(i)$. It follows that j is bounded above by a geometric random variable with mean $O(1)$, as desired. $\qquad\square$

6 Conclusion

We have introduced *bubble-up cuckoo hashing*, a variation of d-ary cuckoo hashing that achieves all of the following properties:

- uses $d = \lceil \ln \epsilon^{-1} + \alpha \rceil$ hash locations per item for an arbitrarily small positive constant α.

- achieves expected insertion time $O(\delta^{-1})$ for any insertion taking place at load factor $1 - \delta \leq 1 - \epsilon$.

- achieves expected positive query time $O(1)$, independent of d and ϵ.

For future work, it would be interesting to try to implement bubble-up cuckoo hashing and compare it implementations of other cuckoo hashing variants as well as other hash table implementations, to find use cases where bubble-up cuckoo hashing may be competitive or superior.

Acknowledgments

References

[1] Yuriy Arbitman, Moni Naor, and Gil Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Automata, Languages and Programming: 36th International Colloquium, ICALP 2009, Rhodes, Greece, July 5-12, 2009, Proceedings, Part I 36*, pages 107–118. Springer, 2009.

[2] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual symposium on foundations of computer science*, pages 787–796. IEEE, 2010.

[3] Tolson Bell and Alan Frieze. $o(1)$ insertion for random walk d-ary cuckoo hashing up to the load threshold. *arXiv preprint arXiv:2401.14394*, 2024.

[4] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. Iceberg hashing: Optimizing many hash-table criteria at once. *Journal of the ACM*, 70(6):1–51, 2023.

[5] Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 182–193. IEEE, 2018.

[6] Julie Anne Cain, Peter Sanders, and Nick Wormald. The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, volume 7, pages 469–476, 2007.

[7] Jie Cui, Jing Zhang, Hong Zhong, and Yan Xu. Spacf: A secure privacy-preserving authentication scheme for vanet with cuckoo filter. *IEEE transactions on vehicular technology*, 66(11):10283–10295, 2017.

[8] Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via xorsat. In *Automata, Languages and Programming: 37th International Colloquium, ICALP 2010, Bordeaux, France, July 6-10, 2010, Proceedings, Part I 37*, pages 213–225. Springer, 2010.

[9] Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science*, 380(1-2):47–68, 2007.

[10] David Eppstein, Michael T Goodrich, Michael Mitzenmacher, and Paweł Pszona. Wear minimization for cuckoo hashing: How not to throw a lot of eggs into one basket. In *Experimental Algorithms: 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29–July 1, 2014. Proceedings 13*, pages 162–173. Springer, 2014.

[11] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[12] Daniel Fernholz and Vijaya Ramachandran. The k-orientability thresholds for g n, p. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 459–468, 2007.

[13] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, 2005.

[14] Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. *Combinatorics, Probability and Computing*, 25(6):870–908, 2016.

[15] Nikolaos Fountoulakis and Konstantinos Panagiotou. Orientability of random hypergraphs and the power of multiple choices. In *International Colloquium on Automata, Languages, and Programming*, pages 348–359. Springer, 2010.

[16] Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the insertion time of cuckoo hashing. *SIAM Journal on Computing*, 42(6):2156–2181, 2013.

[17] Alan Frieze and Tony Johansson. On the insertion time of random walk cuckoo hashing. *Random Structures & Algorithms*, 54(4):721–729, 2019.

[18] Alan Frieze and Páll Melsted. Maximum matchings in random bipartite graphs and the space utilization of cuckoo hash tables. *Random Structures & Algorithms*, 41(3):334–364, 2012.

[19] Alan Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM Journal on Computing*, 40(2):291–308, 2011.

[20] Alan Frieze and Samantha Petti. Balanced allocation through random walk. *Information Processing Letters*, 131:39–43, 2018.

[21] Michael T Goodrich, Daniel S Hirschberg, Michael Mitzenmacher, and Justin Thaler. Cache-oblivious dictionaries and multimaps with negligible failure probability. In *Mediterranean Conference on Algorithms*, pages 203–218. Springer, 2012.

[22] Megha Khosla and Avishek Anand. A faster algorithm for cuckoo insertion and bipartite matching in large graphs. *Algorithmica*, 81(9):3707–3724, 2019.

[23] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[24] Eric Lehman and Rina Panigrahy. 3.5-way cuckoo hashing for the price of 2-and-a-bit. In *Algorithms-ESA 2009: 17th Annual European Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings 17*, pages 671–681. Springer, 2009.

[25] Marc Lelarge. A new approach to the orientation of random hypergraphs. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 251–264. SIAM, 2012.

[26] Xiaozhou Li, David G Andersen, Michael Kaminsky, and Michael J Freedman. Algorithmic improvements for fast concurrent cuckoo hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[27] Colin McDiarmid et al. On the method of bounded differences. *Surveys in combinatorics*, 141(1):148–188, 1989.

[28] Moni Naor, Gil Segev, and Udi Wieder. History-independent cuckoo hashing. In *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*, pages 631–642. Springer, 2008.

[29] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[30] W Wesley Peterson. Addressing for random-access storage. *IBM journal of Research and Development*, 1(2):130–146, 1957.

[31] Stefan Walzer. Insertion time of random walk cuckoo hashing below the peeling threshold. In *30th Annual European Symposium on Algorithms (ESA 2022)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2022.

[32] Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. *ACM Transactions on Algorithms*, 19(3):1–22, 2023.

A   Proof of Proposition 4.1

To prove Proposition 4.1, we begin with the following claim:

Claim A.1. Let $k \leq O(n \log n)$, and let $X_1, X_2, \ldots, X_k$ be iid uniformly random elements of $[n]$. Then, with high probability in n,

$$\Big| \bigcup_i X_i \Big| = n \cdot (1 - e^{-k/n}) \pm O(\sqrt{n} \log n).$$

Proof. The probability that a given $j \in [n]$ is not in any of $X_1, \ldots, X_k$ is

$$(1 - 1/n)^k = e^{-k/n} \pm O(1/n).$$

It follows that

$$\mathbb{E}\left[\Big| \bigcup_i X_i \Big| \right] = n \cdot (1 - e^{-k/n}) \pm O(1).$$

Moreover, if we define $f(X_1, X_2, \ldots, X_k) = |\bigcup_i X_i|$, then f is a function of k independent random variables, each of which affects f's value by at most ± 1 (that is, changing X_i for some i changes f by at most 1). Thus we can apply McDiarmid's inequality (Theorem 3.3) to obtain the concentration bound

$$\Pr[|f - \mathbb{E}[f]| \geq t\sqrt{k}] = e^{-\Omega(t^2)}.$$

Plugging in $t = \Theta(\sqrt{\log n})$ allows us to conclude that, with probability $1 - 1/e^{\Omega(t^2)} = 1 - 1/\operatorname{poly}(n)$, we have

$$\left|\bigcup_i X_i\right| = n \cdot (1 - e^{-k/n}) \pm O\left(t\sqrt{k}\right) = n \cdot \left(1 - e^{-k/n}\right) \pm O\left(\sqrt{n}\log n\right).$$

$\square$

We now prove Proposition 4.1:

Proposition 4.1. Let $\epsilon \in (n^{-1/4}, 1)$. Suppose we sample iid uniformly random coupons $u_1, u_2, \ldots \in [n]$, stopping once we have sampled a total of $(1 - \epsilon)n$ distinct coupons. With probability $1 - 1/\operatorname{poly}(n)$, the number of sampled coupons is

$$n \ln \epsilon^{-1} \pm \tilde{O}(n^{-1/4}).$$

Proof of Proposition 4.1. By Claim A.1, there exists a positive constant c, such that, with high probability in n, we have:

- The first $n \ln(\epsilon^{-1} - c \log n/\sqrt{n})$ coupons sample greater than $(1 - \epsilon)n$ distinct values.

- The first $n \ln(\epsilon^{-1} + c \log n/\sqrt{n})$ coupons sample fewer than $(1 - \epsilon)n$ distinct values.

It follows that the number of coupons needed to sampled $(1 - \epsilon)n$ distinct values is in the range

$$n \ln(\epsilon^{-1} \pm O(\log n/\sqrt{n})).$$

So long as $\epsilon \geq n^{-1/4}$, this range is contained in the range

$$n \ln(\epsilon^{-1} \cdot (1 \pm O(\log n/n^{1/4}))) = n \ln \epsilon^{-1} \pm O(\log n/n^{1/4}).$$

$\square$

B   Implementing $\texttt{choice}(x)$

In the body of the paper, we assume access to a constant-time function $\texttt{choice}(x)$ that, for a given element x, identifies which hash function h_i was used to place x in its current position. In this section, we will show how to remove this assumption while preserving the final bounds that we achieve.

Note that we only ever invoke $\texttt{choice}(x)$ when we already know the current position j containing x. So we can feel free to define $\texttt{choice}(x)$ to actually take two arguments x and j. Our approach in this section will be to replace $\texttt{choice}(x, j)$ with an explicit protocol $\texttt{choice}'(x, j)$ that (with a bit of additional algorithmic case-checking) preserves both the correctness and time guarantees from the main sections of the paper. So that we can discuss both the basic bubble-up algorithm and the advanced bubble-up algorithm simultaneously, we will use $d_{\max}$ to denote d for the basic algorithm and $d_{\max}$ for the advanced

algorithm; we will use d_{core} to mean 2 for the basic algorithm and to mean d_{core} for the advanced algorithm; and we will use ϵ_{core} to mean 0.51 for the basic algorithm and to mean ϵ_{core} for the advanced algorithm.

The protocol $\mathtt{choice}'(x, j)$ examines positions $h_{d_{\max}}(x), h_{d_{\max}-1}(x), \ldots$, one after another, and returns

$$\max\{i \le d_{\max} \mid h_i(x) = j\}. \tag{9}$$

Note that, even though $\mathtt{choice}'(x, j)$ evaluates $h_{d_{\max}}(x), h_{d_{\max}-1}(x), \ldots$, these do not count as probes (and, specifically, first-time probes) in the body of the paper.

To argue that we can use $\mathtt{choice}'$ in place of $\mathtt{choice}$, there are two issues we must be careful about:

- **Issue 1:** $\mathtt{choice}(x, j)$ does not necessarily equal $\mathtt{choice}'(x, j)$. In particular, if there exist $i_1 < i_2 \le d_{\max}$ such that $h_{i_1}(x) = h_{i_2}(x)$, then we could have $\mathtt{choice}(x, j) = i_1$ but $\mathtt{choice}'(x, j) = i_2$.

- **Issue 2:** $\mathtt{choice}'(x, j)$ is not a constant-time function. Thus we must analyze its time contribution to each insertion.

Handling Issue 1. We begin by handling Issue 1, as it is the more significant of the two. Note that, in our algorithms, if $\mathtt{choice}(x, j) > d_{\max} - d_{\mathrm{core}}$, then we do not care about the specific value that $\mathtt{choice}(x, j)$ takes in the range $(d_{\max} - d_{\mathrm{core}}, d_{\max}]$. Thus, the case where Issue 1 can be a problem is if $i_1 \le d_{\max} - d_{\mathrm{core}}$.

Call an element x ***corrupt*** if there exists $i_1 \le d_{\max} - d_{\mathrm{core}}$ and i_2 satisfying $i_1 < i_2 \le \max$ such that $d_{i_1}(x) = d_{i_2}(x)$. These are the elements that, at any given moment, cause Issue 1 to be a problem. It is worth noting that, with high probability, there are very few such elements.

Lemma B.1. At any given moment, we have with probability $1 - 1/\operatorname{poly}(n)$ that there are at most $O(\log^3 n)$ corrupt elements.

Proof. Each element independently has at most an $O(d_{\max}^2/n)$ probability of being corrupt. Thus, by a Chernoff bound, we have with probability $1 - 1/\operatorname{poly}(n)$ that the number of corrupt elements is at most $O(d_{\max}^2 \log n) = O(\log^3 n)$. □

The fact that there are $O(\log^3 n)$ corrupt elements means that they have negligible impact on the various quantities addressed in the analysis. This includes both intermediate quantities used in the analysis (e.g., the total number of non-core-hash-table first-time probes made during some time period) as well as the main quantity that we actually care about (the number of elements in the core hash table). Thus, one can proceed with exactly the same analyses as in the body of the paper in order to achieve the desired bound of $(1 - \epsilon_{\mathrm{core}})n$ on the number of elements in the core hash table.

The other place where Issue 1 shows up is more subtle. Recall that, in order so that we can treat the core hash table as a cuckoo hash table, we need what we call the *core independence property*: that, for each element x, when x gets placed into the core hash table, the hashes $h_{d_{\max}-d_{\mathrm{core}}+1}(x), \ldots, h_{d_{\max}}(x)$ are still fully random (and independent of the state of the core hash table).

A priori, a corrupt element could cause us to break the core independence property as follows. Suppose that $h_i(x) = h_j(x)$ for some $i \le d_{\mathrm{core}} - d_{\max} < j$. When x performs a first-time probe on $h_i(x)$, it will use the position if and only if it is vacant. However, if x does use $h_i(x)$, then in the future, the $\mathtt{choice}'$ function will conclude that x is in the core table. Thus, whether or not the $\mathtt{choice}'$ function in the future perceives x to be in the core table depends on the state of position $h_i(x)$ right now, which depends on the state of the core table right now.

To resolve this issue, we add one final modification to the algorithm: whenever we perform a probe on $h_i(x)$ for some $i \le d_{\max} - d_{\mathrm{core}}$, we also check (in $O(1)$ time) whether $h_i(x) = h_j(x)$ for any

$j \in (d_{\max} - d_{\mathrm{core}}, d_{\max}]$; if such a collision occurs, then we immediately place x into the core table (*without needing to examine the contents of position $h_i(x)$ first*).

This still does not give us the full core-independence property, but it does give us a slightly weaker version of the property: when an element x is placed into the core hash table, the only thing that we have revealed about its core hashes so far is that they are *not* equal to any of the non-core hashes $h_1(x), h_2(x), \ldots, h_{i-1}(x)$ for x that we have performed first-time probes on so far. Another way to think about this is that, when an element x is placed into the core hash table, there is some known set F of $O(\log n)$ *forbidden* hashes (i.e., the non-core hashes that we have already verified are distinct from x's core hashes), and that x's core hashes are uniformly and independently random from the set $[n] \setminus F$. Call this the *weak core-independence property.*

To apply the weak core-independence property, we need slightly *stronger* versions of Theorems 3.1 and 3.2:

Proposition B.2. Let $d = O(1)$. Consider the setups in Theorems 3.1 and 3.2, and suppose that, before each insertion, an adaptive adversary (who gets to see the current hash table state but not the hashes of future elements) is permitted to choose a set of $O(\log n)$ *forbidden hashes* $F \subseteq [n]$ for the next insertion. This means that the next element x to be inserted then has its hashes drawn uniformly and independently random from $[n] \setminus F$. Even with this modification, the conclusions of the theorems continue to hold.

Proposition B.2 tells us that the weak version of the core independence property is still sufficient for us to obtain the desired guarantees from the core hash table. We will defer the proof of the proposition the end of the section. However, assuming that it is true, this completes our solution to Issue 1.

Handling Issue 2. Issue 2 concerns the time spent evaluating the $\mathtt{choice}'$ function on any given insertion.

In both the basic and advanced versions of the bubble-up algorithm, we have the following property: if an element y is part of an eviction chain, and if it is not the *final* member of the eviction chain, then the time spent on y is at least $\Omega(d_{\max} - \mathtt{choice}'(y))$. Thus, with the exception of the final element z in the eviction chain, the $O(d_{\max} - \mathtt{choice}'(x))$ time spent evaluating $\mathtt{choice}'$ can be amortized to the time already spent on the insertion. The final element in the eviction chain spends at most $O(d_{\max})$ time evaluating $\mathtt{choice}'$. For the basic bubble-up algorithm, this adds at most $O(d_{\max}) = O(d) = O(\ln \epsilon^{-1})$ time to each insertion, which is already permitted in Theorem 4.1. For the advanced bubble-up algorithm, if the hash table is at load factor $1 - \delta$, then $d_{\max} = O(\log \delta^{-1}) = O(\delta^{-1})$, so the time spent on the final call to $\mathtt{choice}'$ does not change the asymptotic expected time spent on the insertion.

Proving Proposition B.2. We complete the section by proving Proposition B.2. We begin with an easier proposition:

Proposition B.3. Consider the setups in Theorems 3.1 and 3.2, and suppose that, before each insertion, an adaptive adversary (who gets to see the current hash table state but not the hashes of future elements) gets to remove some subset of the elements and rearrange the remaining elements arbitrarily. Even with this modification, the conclusions of the theorems continue to hold.

Proposition B.3 follows from the the same sequence of arguments as for the original versions of Theorems 3.1 [29] and 3.2 [3]. In both cases, the analysis still works even if the initial arrangement of the elements is arbitrary (indeed, this is explicit in [3]), and even if some elements are omitted by an adaptive adversary.

Using Proposition B.3, we can establish Proposition B.2 as follows:

Proof of Proposition B.2. Suppose that, for the i-th insertion, we have forbidden set F_i. To generate the i-th insertion x_i, we will generate a sequence of elements $x_i^{(1)}, x_i^{(2)}, \ldots$, each of which has fully random hashes, and we will set x_i to be the first one that respects the forbidden set F_i. If $x_i = x_i^{(j)}$ for some

$j > 1$, then we consider $x_i^{(1)}, x_i^{(2)}, \ldots, x_i^{(j-1)}$ to be *phantom elements*. That is, we will imagine that these elements actually *were* part of the insertion sequence, but that an adversary simply chooses to omit them during all future insertions. (This adversary is in the style of the adversary from Proposition B.3.)

Note that phantom elements are extremely rare. Each insertion independently has a probability at most $O(|F|/n) = \tilde{O}(1/n)$ of being a phantom element, so by a Chernoff bound, the total number of phantom elements across all insertions is, with very high probability, at most polylog n. These phantom elements therefore have a negligible overall effect on the total number of insertions that we perform.

Applying Proposition B.3, we can conclude that, even with an adversary omitting the phantom elements, the expected time per insertion remains $O(1)$. This implies Proposition B.2, as desired. □

C Supporting deletions with tombstones

In this section, we describe how to use tombstones to prove the following corollary of Theorem 1.1:

Corollary 3.1. Let $\alpha \in (0, 1)$ be a positive constant, and let $d_{\text{core}} \in O(1)$ be sufficiently large as a function of α. Then, for any $\epsilon \le e^{-d_{\text{core}}}$ satisfying $\epsilon^{-1} \le n^{1/4}$, there exists an implementation of d-ary cuckoo hashing that supports both insertions and deletions and that:

- uses $d = \lceil \ln \epsilon^{-1} + \alpha \rceil$;

- achieves amortized expected time $O(\delta^{-1} \log \delta^{-1})$ for any insertion/deletion taking place at a load factor $1 - \delta \le 1 - \epsilon$;

- achieves expected positive query time $O(1)$.

Proof. We will implement deletions by *marking* the appropriate element as deleted. If an element marked as deleted is later reinserted, it is simply un-marked. Elements that are marked as deleted are referred to as **tombstones**. Tombstones participate in eviction chains just like any other element – from the perspective of insertions, they are regular elements.

Because insertions treat tombstones as elements, one must be careful to limit the **augmented load factor** of the hash table, which is the load factor *including* the tombstones. Whenever the augmented load factor reaches $1 - \epsilon'$, for some $\epsilon' = \Theta(\epsilon)$ to be chosen later, the hash table is rebuilt from scratch and the tombstones are cleared out. These rebuilds occur every $O(\epsilon n)$ operations, and cost

$$\int_{m=0}^{(1-\epsilon)n-1} O(n/(n-m))^{-1} \mathrm{d}m = O(n \log \epsilon^{-1})$$

expected time. The amortized expected rebuild cost per insertion is therefore

$$O\left(\frac{n \log \epsilon^{-1}}{\epsilon n}\right) = O(\epsilon^{-1} \log \epsilon^{-1}).$$

The tombstones raise the maximum load factor that the hash table must support from $1 - \epsilon$ to $1 - \epsilon'$. Setting $\epsilon' = e^\alpha \epsilon$, we can use Theorem 1.1 to get $d = \lceil \ln((\epsilon')^{-1}) + \alpha \rceil = \lceil \ln \epsilon^{-1} + 2\alpha \rceil$. Finally, by using $\alpha/2$ in place of α, we obtain the desired overall bound on d.

Note that, if an insertion failure occurs, we can handle it by just performing an immediate rebuild. The expected number of times that this happens during a given time window between scheduled rebuilds is $n^{-\Omega(1)}$ (by Theorem 1.1), so these extra failure-induced rebuilds contribute negligibly to the overall amortized expected insertion cost. □